

Queries over Overlapping XML Structures

Ionut E. Iacob^{*}
Department of Computer Science
University of Kentucky
Lexington, KY, USA
eiaco0@cs.uky.edu

Alex Dekhtyar[†]
Department of Computer Science
University of Kentucky
Lexington, KY, USA
dekhtyar@cs.uky.edu

ABSTRACT

In recent years it has been argued that when XML encodings become complex, DOM trees are no longer adequate for query processing. Alternative representations of XML documents, such as multi-colored trees have been proposed as a replacement for DOM trees for complex markup. In this paper we consider the use of Generalized Ordered-Descendant Directed Acyclic Graphs (GODDAGs) for the purpose of storing and querying complex document-centric XML. GODDAGs are designed to store multihierarchical XML markup over the shared PCDATA content. They support representation of overlapping markup, which otherwise cannot be represented easily in DOM. We describe how the semantics of XPath axes can be modified to define path expressions over GODDAG, and enhance it with the facilities to traverse and query overlapping markup. We provide efficient algorithms for axis evaluation over GODDAG and describe the implementation of the query processor based on our definitions and algorithms.

1. INTRODUCTION

XML has become a popular approach to storage and transfer of diverse data because of its simplicity and transparency, as well as because of wide availability of (free) tools for working with it. Availability of open-source XML-related standards allows software developers to build XML-enabled applications in a straightforward manner: XML files are parsed using a combination of SAX and DOM parsers, constructed memory-resident DOM trees are accessed from applications via DOM API calls. More complex XML management tasks involve the use of XPath [2] and/or XQuery [3] expressions for querying the content of DOM trees and XSLT for converting the content/structure of the tree, usually, for the purpose of visualizing the data. For simple XML

data, XPath/XQuery over DOM Trees provide efficient and convenient way for querying.

However, straightforward approaches to organizing XML for querying might yield unsatisfactory solutions when complex markup is considered. In [13] Jagadish et al. observed that querying XML data in the presence of several hierarchies for marking up features of the same objects can be done more efficiently if alternative data structures are used in place of a number of independent DOM trees for each of the hierarchies. Jagadish et al. proposed a data structure called *multi-colored tree* (MCT) for storing such markup and discussed efficient query evaluation strategies.

The approach of [13] was designed with data-centric XML in mind. The multicolored tree structure is built on top of individual XML nodes. This allows hierarchies of different “colors” to share content of some of the nodes. When *document-centric* XML is considered, however, there is an additional dimension, not captured by MCTs: the *sharing* of information in the hierarchies occurs at the level of content, rather than XML elements. Indeed, typically, document-centric XML documents are built by starting with a text and introducing various markup *on top* of it. When more than one hierarchy is used to encode features of a text, often the scopes of different markup elements *overlap*. This is illustrated on the following example.

EXAMPLE 1. Consider a fragment of [15], shown in Figure 1. We describe two markup hierarchies for this document. First hierarchy describes, using elements `<p>` (paragraph), `<sentence>` and `<w>` (word), the structure of the text of the document. Our second hierarchy uses elements `<page>` and `<line>` to describe the physical layout of the text. The (somewhat simplified) corresponding XML encodings of this fragment are shown in Figures 2.(a) and 2.(b). Examination of the scopes of the XML elements in these figures reveals numerous overlapping conflicts. In particular, we mention the conflict between the scope of the `<page no="1">` element and the content of both `<p>` and `<sentence no="14">` elements. Similarly, the `<w>` element around the word "fundamental" overlaps both `<line no="1">` and `<line no="2">` elements of `<page no="2">`. Overall, even this simple fragment, described using just two hierarchies contains six pairs of elements with overlapping content.

Overlap in content of elements means that the markup presented in Figure 2 cannot be stored in a single XML document/DOM tree in a straightforward manner. As they lack facilities to store overlapping markup, it also cannot be stored in a single MCT. At the same time, storing each

^{*}Work supported, in part, by the NEH grant RZ-20887-02.

[†]Work supported, in part, by the NSF grant ITR-0219924 and the NSF grant ITR-0325063.

Where there are charges that by one means or another the vote is being denied, we must find out all of the facts -- the extent, the methods, the results. The same is true of substantial charges that

- 2 -

unwarranted economic or other pressures are being applied to deny fundamental rights safeguarded by the Constitution and laws of the United States.

Figure 1: A fragment of a letter detailing the proposed Civil Rights Program to the members of President Eisenhower's Cabinet.

hierarchy in a separate DOM tree is inefficient from the perspective of query processing. For example, a user query

Find all sentences completely or partially located on page 1, which contain the word "charges"

requires navigation through both text structure and physical location markup. Similar to the cases considered in [13], executing such a query as a join is inefficient. To make matters worse, the full answer to this query must include sentence number 14, only partially located on page 1. This means that the abovementioned query is not expressible in XPath (or XQuery, for that matter) over the set of the two encodings in Figure 2.

The goal of this paper is to describe the framework for representing and querying overlapping XML markup from different hierarchies. To researchers in the field of document processing, this problem is known as the problem of management of *concurrent XML* [17, 8, 16]. It has been known since the days of SGML [16], and has found its acknowledgment in the Guidelines of the Text Encoding Initiative (TEI)[17]. To date, solutions proposed in that community concentrate on representation, either using XML syntax [17, 8], or going beyond it [12, 7]. In both cases, query processing is problematic. Computer Scientists started looking at concurrent markup only recently studying both the problem of automated maintenance of multihierarchical documents [6] and on data structures for their representation[14].

In [18], Sperberg-McQueen and Huitfeldt have introduced Generalized Ordered-Descendant Directed Acyclic Graphs (GODDAGs), a data structure for storing concurrent/ multihierarchical markup. A GODDAG combines DOM trees of individual XML hierarchies together by "tying" them at the top, root, level and at the bottom, content level. In [18], Sperberg-McQueen cite the need for appropriate mechanisms for building GODDAGs and querying data stored in them. The former problem had been addressed in [1]. In this paper, we adopt GODDAG (formally described in Section 2) as the data structure for storing concurrent markup. We then proceed to

- define the semantics of XPath axes over multiple hierarchies in GODDAG structures (Section 3);
- enhance XPath syntax and semantics with constructs for capturing overlapping markup (Section 3);
- develop and present efficient algorithms for axis evaluation over GODDAG (Section 4);

- conduct a preliminary study of the efficiency of enhanced XPath over GODDAG as the means of querying multihierarchical, overlapping markup (Section 5).

2. DATA STRUCTURE FOR OVERLAPPING HIERARCHIES

2.1 Overlapping hierarchies: why do we need a new data structure?

As illustrated in Figures 1 and 2, overlapping markup is a reality when dealing with text encodings. The availability of free processing tools makes XML attractive for representing text structure even in this case, even though single DOM Trees are inadequate for direct representation of the desired markup.

TEI Guidelines [17] propose two solutions for representing overlapping hierarchies in the *same* XML document: *milestone elements* and *fragmentation*. Milestones are empty elements that serve as markers of the place (in PCDATA) where a certain markup feature begins. The "scope" of the feature marked up by a milestone element is the PCDATA content between this element and another milestone element, symbolizing the closing tag. For instance, in the example of Figure 2 we can choose to turn `<sentence>` into a milestone element. In this case, each sentence will now start with a tag `<sentence/>`. The end of the sentence can be marked up by a new `<sentenceEnd/>` tag, or, simply by the next `<sentence/>` tag. Use of milestone elements can solve all overlapping conflicts in the markup - whenever the scopes of two elements overlap, one of them is turned into one or two, if needed, milestone elements.

Fragmentation is another technique to avoid markup conflicts. Given two elements with overlapping scopes, it works by splitting one of them into two fragments at the point where the other markup starts (or ends). Special "glue" attribute (or attributes) is (are) used to preserve the information that the two new elements are parts of the original single element. For instance, the conflict of the 14th sentence and pages in Figure 2 can be resolved as follows:

```
<page no="1">...  
  <sentence no="14" key="1">...</sentence>  
</page><page no="2">  
  <sentence no="14" key="1">...</sentence>  
  ...</page>
```

The sentence markup `<sentence no="14">` was split into

<pre> <doc id="CP56483"> ... <p> <sentence no="13"> <w>Where</w> <w>there</w> are <w>charges</w> that by one means of another the vote is being denied, we must find out all of the facts -- the extent, the methods, the results. </sentence> <sentence no="14">The same is true of substantial <w>charges</w> that unwarranted economic of other pressures are being applied to deny <w>fundamental</w> <w>rights</w> <w>safeguarded</w> by the Constitution and laws of the United States. </sentence> </p> ...</doc> </pre>	<pre> <doc id="CP56483"> ... <page no="1"> <line no="31">Where there are charges that by one means of another the vote</line> <line no="32">is being denied, we must find out all of the facts -- the extent, the</line> <line no="33">methods, the results. The same is true of substantial chargers that</line> </page> <page no="2"> <line no="1">unwarranted economic of other pressures are being applied to deny funda</line> <line no="2">mental rights safeguarded by the Constitution and laws of the United</line> <line no="3"> States. ... </page> ...</doc> </pre>
(a)	(b)

Figure 2: Encoding of the fragment from Figure 1: (a) text structure, (b) physical location.

two fragments related by the same value of the glue attribute key.

Both milestones and fragmentation succeed in representing overlapping markup in a single XML document/DOM tree. However, there is a price to pay at the query processing time. In both cases the semantics of the constructed DOM Tree *is no longer the semantics of the underlying document*. Indeed, milestone elements have *no content* in DOM Trees, whereas the XML elements/features they represent in the document encoding *do have content*. Similarly, when fragmentation is used, a single feature becomes represented by more than one XML element, with its content equal to the union of the contents of the fragments. This causes inconvenience at two levels: (a) at the document generation stage and (b) during query processing.

At the document generation stage *human editors* are expected to make decisions about turning elements into milestones or fragmenting them. Methods for automatically doing so are only recently emerging[6].

This paper concentrates on the second inconvenience, namely, query processing over the DOM trees achieved using fragmentation and/or milestones. The issues to be addressed can be illustrated on the following example. Consider the milestones approach for answering the question query from Section 1 using an XPath query. We represent sentences with single milestones, all sentences document will look as follows:

```

<doc id="CP56483"> <page no="1"> ... <p/>
<line no="31"> <sentence no="13"/>
  <w>Where</w> ... vote</line>
<line no="32">is being ... the</line>
<line no="33">methods, the results.
  <sentence no="14"/>The same ...
  <w>charges</w> that</line>
</page>
<page no="2">
<line no="1"> unwarranted ... deny
  <w/>funda </line>
<line="2">mental<wEnd/> ... United</line>
<line="3">States.<sentenceEnd/></line> ...
</page> ...</doc>

```

We observe, that it is relatively easy to obtain the `<sentence>` elements that are located inside page 1, the XPath query `//page[@no="1"]/sentence` will produce the necessary data. However, this is not the entire query - it is also possible, that a sentence starts on a previous page and continues on the

current one. To capture all sentence elements we need to use a more complex XPath expression

```

//sentence[ancestor::page[@no="1"] or
  following::sentence[1][ancestor::page[@no="1"]]]

```

Next step is to establish which sentences contain the word “charges”. If `<sentence>` had scope, we would have written the following XPath expression `//sentence[descendant::w[string(.)="charges"]]`. Because `<sentence>` is a milestone, we must, upon discovering the word “charges” in text, find the previous (preceding) occurrence of `<sentence>`. Combined with the expression above (taking into account, that the word must be “inside” the sentence), we get

```

//w[string(.)="charges"]
/preceding::sentence[1][ancestor::page[@no="1"] or
  following::sentence[1][ancestor::page[@no="1"]]]

```

This query makes a simplifying assumption that the word “charges” is not split in the document (imagine how our search would be complicated if we want to find all sentences which contain the word “fundamental” on page 2). Another issue is that this expression will produce a set of empty nodes. If we want to obtain the actual *text* of the sentences, we are looking at significantly more work. In particular, we must now collect the PCData content for each node in the DOM tree between the milestone `<sentence>` element returned as the answer to the query above, and the next milestone `<sentence>` or `<sentenceEnd>` element. This process *cannot be represented by an XPath 1.0 expression*.

An attempt to build an XPath expression for the same query over the DOM tree obtained as a result of using fragmentation leads to similar complications. We can start with `//sentence[ancestor::page[@no="1"] or descendant::page[@no="1"]]`

`[descendant::w[string(.)="charges"]]` which will return all sentence fragments that are completely within page 1. There are at least two problems with this expression: (i) it fails to deliver sentences that overlap page 1 *but contain the word “charges” on another page* ¹ (ii) it delivers only *sentence fragments* instead of full sentences.

¹Consider, for example, searching for sentences on page 1 that contain the word “Constitution”.

We can remove problem (i) as follows (we consider only including the fragments of a sentence that ends *after* page 1):

```
//sentence[ancestor::page[@no="1"] or
  descendant::page[@no="1"] or
  (preceding::page[@no="1"] and string(@id)=
  string(preceding::sentence
    [ancestor::page[@no="1"] or
    descendant::page[@no="1"]]/@id))]
  [descendant::w[string(.)="charges"]]
```

Problem (ii), however, cannot be completely solved using only XPath 1.0 expressions. We must also observe that queries involving node position are particularly difficult to be answered using fragmentation model.

The use of both milestones and fragmentation results in an *approximate representation* of the document structure in a single DOM tree. As shown in the examples above, even the most simple queries cannot be fully evaluated solely by XPath processors. In large part, this is due to the severe restrictions on how structures can be represented by DOM trees. In the following subsection we try to overcome the problems of milestones and fragmentation and we propose a data structure that precisely represents overlapping structures and allows expressing queries over overlapping hierarchies.

2.2 Data structure for representing overlapping hierarchies

We identify three basic principles for choosing a data structure for overlapping hierarchies: (i) we want to preserve individual hierarchies inside the complete document representation, (ii) we want to easily navigate from one structure to another, and (iii) we want to capture relationships between elements in different hierarchies.

It is a fact that complex queries are likely to expensive ([11, 10]). In [13] it is pointed out that, even for complex hierarchies, a tree-like structure is desirable due its relative navigation simplicity.

We start by introducing *concurrent markup hierarchies* and *distributed XML documents*. A concurrent markup hierarchy (CMH) is a collection of schema definitions (DTDs, X Schemas, etc. . .) that share a single (root) element name, and no other element names². Individual schemas are called *hierarchies*. Given a CMH $C = \langle T_1, \dots, T_k \rangle$, a distributed XML document (DXD) D over C is a collection of XML documents (d_1, \dots, d_k) , one for each hierarchy of C , such that all documents **have the same PCDATA content**. Individual documents d_i are called *components* of D . They are not expected to be valid w.r.t. their schema T_i , but must contain markup only from T_i . This separation of markup in a DXD addresses principle (i) above: each document preserves a structure. Two XML documents in Figure 2 show us an example of a DXD with two document components: d_1 on top (corresponding to a “text” hierarchy), and d_2 at the bottom (corresponding to a “physical layout” hierarchy). As clear from this example, DXDs can incorporate within them overlapping markup.

Representing DXD components as individual independent DOM trees is inconvenient, as illustrated in [13]. Instead, we use a structure called *General Ordered-Descendant Directed Acyclic Graph (GODDAG)*, originally introduced by

²Namespaces can be used to distinguish elements from different hierarchies with the same name, but this fact is not important for the scope of this paper.

Sperberg-McQueen and Huitfeldt in [18] precisely for the purpose of storing concurrent markup. Informally, a GODDAG for a distributed XML document D can be thought of as the graph that unites the DOM trees of individual components of D , by merging the root node and the text (PCDATA). Because of possible overlap in the scopes of XML elements (text nodes) from different component documents, the underlying content of the document is stored *not* in text nodes, but in a special *new type* of node called *leaf node*. In a GODDAG, leaf nodes are children of the text nodes, and they represent a consecutive sequence of content characters that is *not broken by an XML tag from any* of the components of the distributed XML document. While each component of D will has its own text nodes in a GODDAG, the leaf nodes will be shared among all of them. As a consequence, **leaf nodes have multiple parents**: one in each component of D .

A GODDAG structure for the DXD in Figure 2 is illustrated in Figure 3. In the figure, nodes in the “text” hierarchy are on the top part, whereas nodes for the “physical layout” hierarchy are at the bottom. Leaf nodes are represented in the middle as rectangles corresponding to the PCDATA they cover. Element nodes are explicitly drawn with names and attribute values. Text nodes are symbolized by T in a circle. To easily identify the nodes, we put a unique label next to each node. Note here, for example, that the word “fundamental” is broken into two leaf nodes: L12: “funda” and L13: “mental”. This allows us to represent the content of the appropriate $\langle w \rangle$ element (116) in the first hierarchy as $\{L12, L13\}$, while including L12 and L13 in the scope of two different $\langle \text{line} \rangle$ elements (29 and 211 respectively).

To define GODDAG formally, we need to introduce some notation. For an XML document d we let $root(d)$ denote the root element of d and $nodes(d)$ – the set of all nodes in DOM of d . For a node $x \in nodes(d)$ we let $string(x)$ be the PCDATA content of x (as defined in XPath [2]). We also set $start, end : nodes(d) \rightarrow \mathbb{N}$ to return the offset positions in $string(root(d))$ of *start tag* and *end tag* respectively for a node $x \in nodes(d)$. If x is a text node, then $start(x)$, $end(x)$ denote the start offset and end offset respectively. For a distributed document D we let $leaves(D)$ represent the set of all *leaf nodes* in D and we extend the domain of functions $string$, $start$, and end over the $leaves(D)$ set. For *leaf nodes* these functions are defined in the same way as for *text nodes*. We define two new functions, $first-leaf, last-leaf : nodes(D) \rightarrow leaves(D)$. Given a node x , these functions return the leftmost and the rightmost (respectively) leaf nodes in the subtree of x . If $string(x) = \epsilon$, then $first-leaf(x)$, $last-leaf(x)$ return the first following (respectively the first preceding), in reverse document order, *leaf node* for x (or *NIL* if such nodes do not exist).

DEFINITION 1. Let $D = (d_1, \dots, d_k)$ be a distributed XML document. A GODDAG of D is a directed acyclic graph (N, E) where the sets of nodes N and edges E are defined as follows:

- $N = \cup_{i=1}^k nodes(d_i) \cup leaves(D)$
- $E = \cup_{i=1}^k \{ (x, y) \mid x, y \in nodes(d_i) \wedge x \text{ is the parent of } y \} \cup \cup_{i=1}^k \{ (x, y) \mid x \in nodes(d_i) \text{ is a text node, } y \in leaves(D) \wedge start(x) \leq start(y) < end(y) \leq end(x) \}$

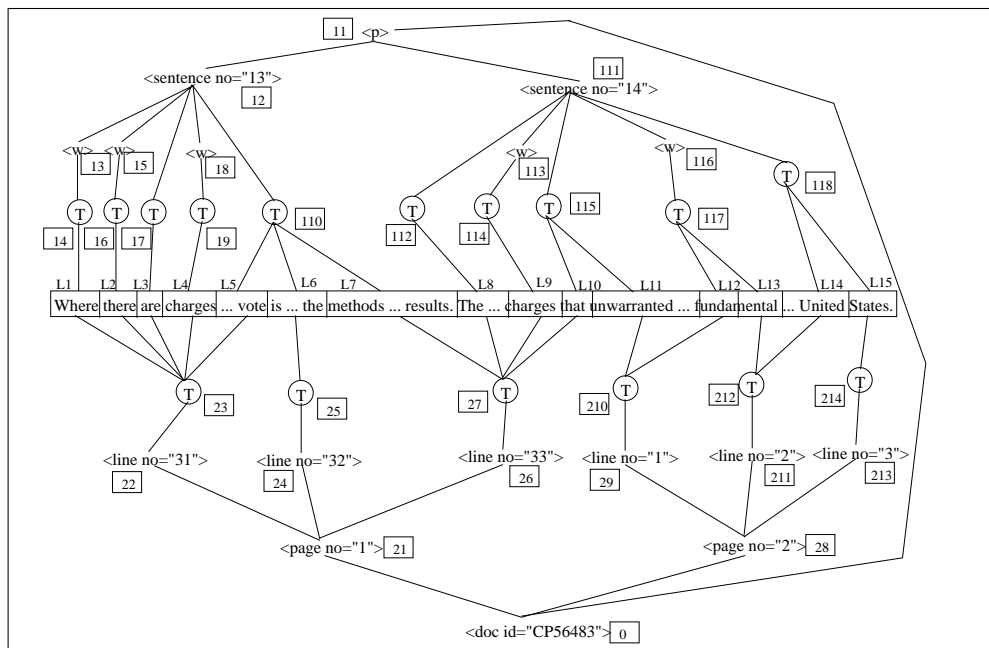


Figure 3: A GODDAG for the distributed document *DXD* in Figure 2

A GODDAG data structure solves nicely the problem of navigation between CMH structures (principle (ii) at the beginning of current subsection): all hierarchies are connected through a common root node and common leaf nodes. The data structure also captures relationships among features in different structures. For instance, in the GODDAG in Figure 3, we can find all sentences partially or totally located on page 1: from `<page no="1">` (node 21) we navigate down and find all leaf nodes it contains (leaves L1 to L10); then we navigate up in the other hierarchy and find all `<sentence>` ancestors for these leaf nodes (nodes 13 and 14).

3. QUERYING DISTRIBUTED XML DOCUMENTS

XPath is a language for addressing parts of an XML document [2]. Although it was initially designed to be used by XSLT and XPointer, XPath is intensively used as part of some XML query languages (XQuery), and can be used itself to query XML documents. In fact, in XQuery queries, XPath expressions are responsible for traversing the underlying XML document model (DOM tree) to discover requested XML nodes.

As shown in Section 2.1, even simple queries are hard to express in XPath over representations of concurrent hierarchies that involve milestone elements and/or fragmentation. In this section we show that when distributed XML documents are represented in GODDAG structures, we can express such queries as path expressions in straightforward ways. In addition, we show that individual components of path expressions (we concentrate on axes) have natural semantics over GODDAG, a semantics that specializes to XPath over DOM semantics when single-component documents are considered. We start by discussing how individual XPath axes can be defined in GODDAGs. After that, we introduce formal definitions of XPath components over

GODDAG and discuss the evaluation of path expressions.

3.1 Path Expressions Over GODDAG

Recall that XPath uses a tree of nodes model to represent an XML document. There are seven types of nodes, the *root node* (a unique node in an XML document), *element*, *text*, *attribute*, *namespace*, *processing-instruction*, and *comment* nodes. The main syntactical construction of XPath is *expression*. An *expression* operates on a *context node* and manipulates *objects* of four kinds: node-set, boolean, string, and numeric.

The instrument for addressing sets of nodes in a document is the *location path* composed of one or more *steps*. Each step consists of an *axis*, a *nodetest* and zero or more *predicates*. An axis determines the direction of traversal from the current (context) node, while nodetests and predicates filter nodes that do not match them. A *location path* syntax can be summarized as follows (comprehensive syntax is given in [2]):

```
locationPath := step1/step2/.../stepn
step := axis::node-test predicate*
predicate := [expression]
```

The main syntactical construction for a *step* evaluation is *axis*: for each node in the current *context node* set an *axis* is evaluated to a set of nodes according to the respective *axis* definition. The set of nodes from *axis* evaluation is filtered by the *node-test* (basically a node type test or a name test for *element* nodes) and *expression* result (evaluated to *true* or *false*) in the context of each node of *axis* evaluation set (*axis* plays the selection role, *node-test* and *predicate* play the filtering role). XPath uses 13 *axes* to address nodes in a document: *ancestor*, *ancestor-or-self*, *attribute*, *child*, *descendant*, *descendant-or-self*, *following*, *following-sibling*, *namespace*, *parent*, *preceding*, *preceding-sibling*, and *self*. Formal semantics for XPath axes is given in [2] as well as in [19, 9].

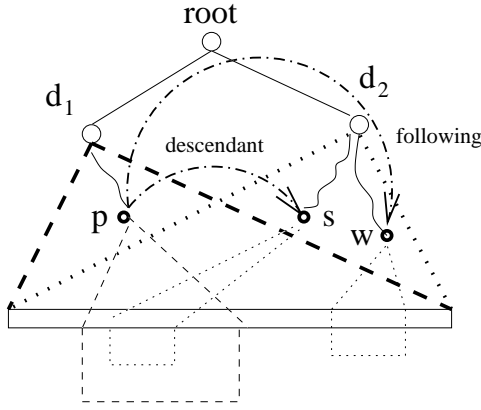


Figure 4: Descendant and following axes in GODDAG.

We illustrate the problem of definition of XPath axes over GODDAG on the following examples. In the context of our example from Figures 1 and 2, consider the query “Find all sentences completely inside page 1”. If `<page>` and `<sentence>` markup were in the same hierarchy, we would have expressed this query using the following XPath expression:

```
//page[@no="1"]/descendant::sentence
```

But in our GODDAG (Figure 3), they are not. Yet, the representation mechanism should not affect our understanding of the relationship between pages and sentences. By definition of the *descendant* axis, a `<sentence>` node is a descendant of a `<page>` node if it is located in the `<page>` node’s subtree. However, we can describe a descendant relationship in a different way:

a node x is a descendant of node y iff the content of x is *completely subsumed* by the content of y .

When considered over DOM trees, these two definitions are (almost) equivalent. The key difference between them is that while the former definition is DOM-specific, the latter is not. In Figure 4 we show how the latter definition can be applied to GODDAG. Here, two components d_1 and d_2 of a DXD are shown. Node p in component d_1 has content that subsumes completely the content of node s from component d_2 . By applying the definition above, we can state that s is a *descendant* of p . Similarly, because the *ancestor* relationship is the inverse of the descendant, we can use the same idea to state that p is an *ancestor* of s in the GODDAG.

We can use similar intuition to redefine *following* and *preceding* axes. Indeed, a node x follows a node y iff the entire PCDATA content of x is located *after* the entire PCDATA content of y in a DOM tree. This statement is, again, independent on the DOM tree structure (as opposed to the definition of the *following* axis, which relies on the document order), and therefore can be transferred to GODDAG, as illustrated in the Figure 4. Node w of component d_2 has content that lies *after* the content of node p , hence, we can state that w *follows* p and, conversely, p *precedes* w .

At the same time, not all axes can be redefined in such a way, in particular, *child* and *parent* cannot transcend the boundaries of a single component in a way *descendant* and *ancestor* do. This is because unlike the notion of descendant,

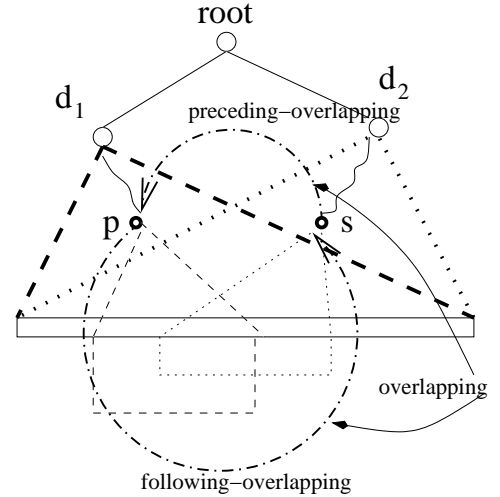


Figure 5: Axes for overlap in GODDAG.

childhood-parenthood relations are tied to tree structures: a node x is a child of a node y iff there is an edge from y to x . Similarly, *preceding-sibling* and *following-sibling* axes rely on the existence of edges between the nodes in the DOM tree, not just on the position of the content. These axes, as well as *self* will not be extended beyond individual components of the distributed document.

One more observation can be made. Given a node x of a DOM tree, the five axes *ancestor*, *descendant*, *self*, *preceding* and *following* partition the entire DOM tree into five disjoint sets of nodes: that is, every node in the DOM tree will belong to exactly one of these axes as traversed from x . This property, however, does not hold in GODDAG: as shown in Section 2, there are GODDAG nodes with *overlapping* content (See Figure 5). Traversing the GODDAG using any of the five axes above will never yield any node that overlaps the context node in content. At the same time, as have been illustrated, queries over GODDAG require computing overlap. To accommodate for this need, we consider enhancing XPath with three new axes: *preceding-overlapping*, *following-overlapping* and *overlapping*. Intuitive meaning of these axes, as illustrated in Figure 5 is quite straightforward: x is in the result of applying *preceding-overlapping* axis to y iff x and y overlap in scope and x starts before y . In this case, y will be in the result of *following-overlapping* applied to x . The *overlapping* axis is the union of *preceding-overlapping* and *following-overlapping*.

We can now proceed to give formal definitions to XPath components over GODDAG, including the enhances apparatus to support markup overlap.

3.2 XPath over GODDAG

Let D be a distributed XML document over a concurrent XML hierarchy $C = \langle T_1, \dots, T_k \rangle$. We define 11 new XPath axes, over the distributed document D , in the context of a node $x \in nodes(D)$: *xancestor*, *xdescendant*, *xancestor-or-self*, *xdescendant-or-self*, *xfollowing*, *xpreceding*, *following-overlapping*, *preceding-overlapping*, *overlapping*, *xancestor-or-overlapping*, and *xdescendant-or-overlapping*. The first six axes are versions of the corresponding XPath axes extended to GODDAG. The remaining five axes do not have

analogs in XPath.

Following the semantics of XPath, for each new axis, \mathcal{X} , we define the corresponding evaluation function

$$\mathcal{X} : nodes(D) \rightarrow 2^{nodes(D)}$$

where $\mathcal{X}(x)$ returns the set of nodes corresponding to axis \mathcal{X} evaluated for the context of node $x \in nodes(D)$.

Xancestor/*xdescendant* axes are defined using superset/subset relation on the content of the nodes, represented via a set of *leaf nodes* in the GODDAG. To define *xfollowing* and *xpreceding* axes, we extend the document order onto the GODDAG. However, the document order over a GODDAG will no longer be total: overlapping markup will be incomparable.

DEFINITION 2. *The following new axes are defined:*

1. *xancestor* ::= $ancestor(x) \cup \{y \in nodes(D - doc_D(x)) \mid start(y) \leq start(x) \leq end(x) \leq end(y)\}$.
2. *xdescendant* ::= $descendant(x) \cup \{y \in nodes(D - doc_D(x)) \mid start(x) \leq start(y) \leq end(y) \leq end(x)\}$.
3. *xancestor-or-self* ::= $xancestor(x) \cup \{x\}$.
4. *xdescendant-or-self* ::= $xdescendant(x) \cup \{x\}$.
5. *xfollowing* ::= $following(x) \cup \{y \in nodes(D - doc_D(x)) \mid start(y) \geq end(x)\}$.
6. *xpreceding* ::= $preceding(x) \cup \{y \in nodes(D - doc_D(x)) \mid end(y) \leq start(x)\}$.
7. *following-overlapping* ::= $\{y \in nodes(D) \mid start(x) < start(y) < end(x) < end(y)\}$.
8. *preceding-overlapping* ::= $\{y \in nodes(D) \mid start(y) < start(x) < end(y) < end(x)\}$.
9. *overlapping* ::= $following-overlapping(x) \cup preceding-overlapping(x)$.
10. *xancestor-or-overlapping* ::= $xancestor(x) \cup overlapping(x)$.
11. *xdescendant-or-overlapping* ::= $xdescendant(x) \cup overlapping(x)$.

We give some examples of the extended axes for the GODDAG shown in Figure 3. (we use node labels to identify nodes in the graph)

(A) *xdescendant*(21) = {22, 24, 26, 23, 25, 27, 14, 16, 17, 19, 110, 13, 15, 18, 12, 112, 114, 113}.

Note 21 corresponds to the `<page no="1">` markup. The *xdescendants* of this node are all its descendants in the “physical layout” component (lines 31, 32 and 33 and corresponding text nodes) as well as the contents of sentence 13 (nodes 12,13,15,18 and the corresponding text nodes). In addition, parts of sentence 14 (node 113 and text nodes 112 and 114) also are *xdescendants* of 21. At the same time, sentence 14 itself is *not* an *xdescendant* of page 1.

(B) *following-overlapping*(26) = {111, 115}

Node 26 represents `<line no="33">` markup from page 1. The scope of its content is leaf nodes L7—L10. The scope of node 111 (`<sentence no="14">`) is L8—L15: because it starts after the scope of node 26 starts and ends after the

scope of node 26 ends, it belongs to the result of evaluation of the *following-overlapping* axis. Incidentally, text node 115 also overlaps node 26 on the right, thus it is added to the result as well.

Remark. We note that Definition 2 allows a node x to be both an *xdescendant* and an *xancestor* of a node y : if $start(x) = start(y)$, $end(x) = end(y)$ and they are in different documents.

Proposed axes allow us to express queries to multihierarchical (distributed) documents in a straightforward manner. Consider, for example, the following queries:

(Q1): Find all sentences completely located on page 2;

(Q2): Find all words located on two lines;

(Q3): Find all sentences completely or partially located on page 1 of the document, that contain the word “charges”;

(Q4): Find all occurrences of the word “Constitution” after page 1.

Table 1 shows the path expressions for these queries.

We can now proceed to describe the evaluation of the newly defined axes.

3.3 Evaluation of extended XPath axes

Let D be a distributed XML document and \mathcal{X} – an extended XPath axis. We define extended axis restriction to each component d_i of D as

$$\begin{aligned} \mathcal{X} : nodes(D) \times docs(D) &\rightarrow 2^{nodes(d_j)} \\ \mathcal{X}(x, d_j) &= \mathcal{X}(x) \cap nodes(d_j), \quad 1 \leq j \leq k. \end{aligned}$$

The axes *xancestor*, *xdescendant*, *xfollowing*, *xpreceding*, *xancestor-or-self* and *xdescendant-or-self* extend the semantics of their counterparts in regular XPath. We say that such an extended axis \mathcal{X} is dual to its respective regular XPath axis \mathcal{A} and we denote this by $\mathcal{A} = dual(\mathcal{X})$. The following results follow directly from Definition 2 and show that the first six extended axes *specialize* to the semantics of their duals.

THEOREM 1. *Let \mathcal{X} denote one of the axes in Definition 2(1-6) and let $\mathcal{A} = dual(\mathcal{X})$. Then for a distributed XML document $D = \langle d_1, \dots, d_k \rangle$ and $\forall x \in nodes(D)$,*

$$\mathcal{X}(x, doc_D(x)) = \mathcal{A}(x)$$

COROLLARY 1. *Let \mathcal{X} denote one of the axes in Definition 2(1-6) and let $\mathcal{A} = dual(\mathcal{X})$. Then for a distributed XML document $D = \langle d \rangle$ and $\forall x \in nodes(D)$,*

$$\mathcal{X}(x) = \mathcal{A}(x)$$

An extended XPath axis is evaluated for each node in the *union of node-sets over the distributed document* and a *union of node-sets* corresponding to the evaluation is returned. Each node in the returned set of nodes constitutes the context node for the (possible) subsequent axis. In Ta-

$\mathcal{S} : Axis \rightarrow NodeSet \rightarrow NodeSet$
$\mathcal{S}[\mathcal{X}](N_1 \cup \dots \cup N_k) = \bigcup_{i=1}^k \mathcal{S}[\mathcal{X}](N_i)$
$\mathcal{S}[\mathcal{X}](N_i) = \bigcup_{j=1}^k \bigcup_{x \in N_i} \mathcal{X}(x, d_j)$

Table 2: Extended XPath axes evaluation

ble 2 we capture the semantics of the extended XPath axis evaluation, starting with a set of context nodes. However, a straightforward implementation of the semantics in Table 2 (for each node in the input set, evaluate the axis by visiting

Query	Path Expression
Q1	/xdescendant::page[@no="2"]/xdescendant::sentence
Q2	/xdescendant::word[overlapping::line]
Q3	/xdescendant::page[@no="1"]/xdescendant-or-overlapping::sentence[descendant::w[string(.)="charges"]]
Q4	/xdescendant::page[@no="1"]/xfollowing::w[string(.)="Constitution"]

Table 1: Using newly defined axes to express queries over multihierarchical XML documents.

all nodes in the target document) would lead to a quadratic time complexity in the size of nodes in D .

Before proceeding further, in order to precisely define a *step* evaluation, we need to give the semantics for a *predicate* evaluation. A *predicate* (and consequently an *expression*) is evaluated in a context of a node in a current node-set. An extended XPath axis evaluation holds a *union* of node-sets over the distributed document. The semantics of a predicate p evaluation for a node in the context of union of node-sets over a distributed document is summarized in Table 3.

$S : Predicate \rightarrow (Node, \cup_1^k NodeSet) \rightarrow boolean$
$S[P](x, N_1 \cup \dots \cup N_k) = S[P](x, N_j), x \in N_j$

Table 3: Extended XPath axes evaluation

Using the semantics in Table 3 it makes sense to use a function like *position()* which is strictly related to the document order (no total document order is defined over a distributed XML document).

Let $1 \leq i, j \leq k, i \neq j, x_1, x_2 \in nodes(d_i), x_1 < x_2$. The following theorems establish important results about how the order of nodes in an axis evaluation (for a node $x \in d_i$ in a *target document* d_j) is related to the order of nodes in the input set of nodes.

THEOREM 2. Let \mathcal{X} denote the xancestor axis.

- $\forall y_1 \in \mathcal{X}(x_1, d_j), \forall z \in nodes(d_j), start(z) > start(x_1) \vee start(x_1) \leq start(z) < end(z) < end(x_1)$ then $y_1 < z$.
- If $start(x_1) < start(x_2), \forall y_1 \in \mathcal{X}(x_1, d_j), \forall y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j)$ then $y_1 < y_2$.
- If $start(x_1) = start(x_2) < end(x_2) \leq end(x_1)$, then $\mathcal{X}(x_1, d_j) \subseteq \mathcal{X}(x_2, d_j)$.
- If $start(x_1) = end(x_1)$, then for $\forall x \in nodes(d_i), start(x_1) = A(y)$ we have $\mathcal{X}(x_1, d_j) \supseteq \mathcal{X}(x, d_j)$.

PROOF. (1) We have $start(y_1) < start(z)$ or $start(y_1) \leq start(x_1) \leq start(z) < end(z) < end(x_1) \leq end(y_1)$, hence $y_1 < z$.

(2) We have $start(y_1) \leq start(x_1)$ and $start(x_1) \leq start(y_2) \leq start(x_1)$. If $start(x_1) < start(y_2)$ we are done: $start(y_1) < start(y_2)$ hence $y_1 < y_2$. If $start(x_1) = start(y_2)$ then we must have $end(y_2) < end(x_1) \leq end(y_1)$. It follows also that $y_1 < y_2$.

(3) Let $y_1 \in \mathcal{X}(x_1, d_j)$. We have $start(y_1) \leq start(x_1) = start(x_2) < end(x_2) \leq end(x_1) \leq end(y_1)$. Hence $y_1 \in \mathcal{X}(x_2, d_j)$, therefore $\mathcal{X}(x_1, d_j) \subseteq \mathcal{X}(x_2, d_j)$.

(4) Let $y \in \mathcal{X}(x, d_j)$. We have $start(y) \leq start(x) = start(x_1) = end(x_1) \leq end(y)$. It follows that $y \in \mathcal{X}(x_1, d_j)$ hence $\mathcal{X}(x_1, d_j) \supseteq \mathcal{X}(x, d_j)$. \square

THEOREM 3. Let \mathcal{X} denote the xdescendant axis.

- $\forall y_1 \in \mathcal{X}(x_1, d_j), \forall z \in nodes(d_j), end(x_1) < start(z)$ then $y_1 < z$.
- If $start(x_1) \leq start(x_2) \leq end(x_2) \leq end(x_1)$ then $\mathcal{X}(x_1, d_j) \supseteq \mathcal{X}(x_2, d_j)$.
- If $end(x_1) = start(x_2)$ then $\mathcal{X}(x_1, d_j) \cap \mathcal{X}(x_2, d_j) = \{y \in nodes(d_j) : end(x_1) = start(y) = end(y) = start(x_2)\}$ and $\forall y_1 \in \mathcal{X}(x_1, d_j) - \mathcal{X}(x_2, d_j), \forall y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j)$ then $y_1 < y_2$.
- If $end(x_1) < start(x_2), \forall y_1 \in \mathcal{X}(x_1, d_j), \forall y_2 \in \mathcal{X}(x_2, d_j)$ then $y_1 < y_2$.

PROOF. (1) We have $start(y_1) \leq end(x_1) < start(z)$. Hence $y_1 < z$.

(2) $\forall y \in \mathcal{X}(x_2, d_j)$ we have $start(x_1) \leq start(x_2) \leq start(y) \leq end(y) \leq end(x_2) \leq end(x_1)$. It follows that $y \in \mathcal{X}(x_1, d_j)$, thus $\mathcal{X}(x_1, d_j) \supseteq \mathcal{X}(x_2, d_j)$.

(3) $\forall y \in \mathcal{X}(x_1, d_j) \cap \mathcal{X}(x_2, d_j), y \neq NIL$ we have $start(x_1) \leq start(y) \leq end(x_1)$ and $start(x_2) \leq start(y) \leq end(x_2)$. It follows that $end(x_1) = start(y) = start(x_2)$. Similarly we get $end(x_1) = end(y) = start(x_2)$. Conversely, for $y \in nodes(d_j), end(x_1) = start(y) = end(y) = start(x_2)$ it follows that $start(x_1) \leq start(y) \leq end(y) \leq end(x_1)$ and $start(x_2) \leq start(y) \leq end(y) \leq end(x_2)$, hence $y \in \mathcal{X}(x_1, d_j) \cap \mathcal{X}(x_2, d_j)$.

Suppose that $\mathcal{X}(x_1, d_j) - \mathcal{X}(x_2, d_j) \neq \emptyset, \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j) \neq \emptyset$. Then $start(y_1) \leq end(y_1) \leq end(x_1) \leq start(x_2) \leq start(y_2) \leq end(y_2)$. Moreover, $start(x_2) < end(y_2)$. Then *start* and *end tags* of y_2 must come after the *end tag* of y_1 , so $y_1 < y_2$.

(4) We have $start(y_1) \leq end(x_1) < start(x_2) \leq start(y_2)$, hence $y_1 < y_2$. \square

THEOREM 4. Let \mathcal{X} denote the xfollowing axis and \mathcal{A} denote the following axis. Let $x \in nodes(d_i), 1 \leq i, j \leq k, i \neq j$, and let $y = parent(d_j, last-leaf(x))$. Then $\mathcal{X}(x, d_j) = \mathcal{A}(y)$.

PROOF. Since $y = parent(d_j, last-leaf(x))$, then $end(x) \leq end(y)$.

Let $z \in \mathcal{A}(y)$. Then $end(y) \leq start(z)$. It follows that $end(x) \leq start(z)$, hence $z \in \mathcal{X}(x, d_j)$.

Conversely, let $z \in \mathcal{X}(x, d_j)$. Then $end(x) \leq start(z)$, so $start(y) < start(z)$. Since y is a parent of a *leaf* node, it must be a text node. So z is not a descendant of y . It follows that $z \in \mathcal{A}(y)$.

This proves that $\mathcal{X}(x, d_j) = \mathcal{A}(y)$. \square

THEOREM 5. Let \mathcal{X} denote the xpreceding axis and \mathcal{A} denote the preceding axis. Let $x \in nodes(d_i), 1 \leq i, j \leq k, i \neq j$, and let $y = parent(d_j, first-leaf(x))$. Then $\mathcal{X}(x, d_j) = \mathcal{A}(y)$.

Theorems 4 and 5 establish that *xfollowing* and *xpreceding* axes can be computed from *following* and *preceding* respectively, for appropriate context nodes in the target document.

For a node $x \in \text{nodes}(d_i)$ we define the *test nodes set* of x for the *following-overlapping* axis \mathcal{X} in d_j as $\hat{\mathcal{X}}(x, d_j) = \mathcal{A}(\text{parent}(d_j, \text{last-leaf}(x_1))) \cap \{z \in d_j \mid \text{start}(x) < \text{start}(z) < \text{end}(x) \leq \text{end}(z)\}$.

THEOREM 6. *Let \mathcal{X} denote the following-overlapping axis and \mathcal{A} denote the ancestor-or-self axis.*

1. $\mathcal{X}(x_1, d_j) \subseteq \hat{\mathcal{X}}(x_1, d_j)$.
2. If $\text{start}(x_1) \leq \text{start}(x_2) < \text{end}(x_2) = \text{end}(x_1)$ then $\hat{\mathcal{X}}(x_1, d_j) \supseteq \hat{\mathcal{X}}(x_2, d_j)$.
3. If $\text{end}(x_1) \leq \text{start}(x_2)$ then $\forall y_1 \in \mathcal{X}(x_1, d_j), \forall y_2 \in \mathcal{X}(x_2, d_j)$ we have $y_1 < y_2$ and $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.
4. If $\text{end}(x_2) < \text{end}(x_1)$ then $\forall y_1 \in \mathcal{X}(x_1, d_j) \cap \mathcal{X}(x_2, d_j), \forall y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j)$ then $y_1 < y_2$.

PROOF. (1) Let $y_1 \in \mathcal{X}(x_1, d_j)$, and $p = \text{parent}(d_j, \text{last-leaf}(x_1))$. We have that $\text{start}(x_1) < \text{start}(y_1) < \text{end}(x_1) < \text{end}(y_1)$ and $\text{start}(p) < \text{end}(x_1) \leq \text{end}(p)$. Hence $y_1 \notin \text{following}(p)$ and $p \notin \text{following}(y_1)$. Since p has no descendants (except for leaf nodes), it follows that p is a descendant of y_1 . From Definition 2 it follows straight forward that $y_1 \in \{z \in d_j \mid \text{start}(x_1) < \text{start}(z) < \text{end}(x_1) \leq \text{end}(z)\}$. Hence $\mathcal{X}(x_1, d_j) \subseteq \hat{\mathcal{X}}(x_1, d_j)$.

(2) Let $y_2 \in \hat{\mathcal{X}}(x_2, d_j)$. We have $\text{start}(x_1) \leq \text{start}(x_2) < \text{start}(y_2) < \text{end}(x_2) = \text{end}(x_1) \leq \text{end}(y_2)$. It follows that $y_2 \in \hat{\mathcal{X}}(x_1, d_j)$, hence $\hat{\mathcal{X}}(x_1, d_j) \supseteq \hat{\mathcal{X}}(x_2, d_j)$.

(3) We have $\text{start}(x_2) < \text{start}(y_1)$ and $\text{start}(y_1) < \text{end}(x_1) \leq \text{start}(x_2)$. Hence $y_1 < y_2$.

Let $z_1 \in \hat{\mathcal{X}}(x_1, d_j)$. Then $\text{start}(z_1) < \text{end}(x_1) \leq \text{start}(x_2)$. So $z_2 \notin \hat{\mathcal{X}}(x_1, d_j)$ and therefore $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.

(4) We have that $\text{start}(y_1) < \text{end}(x_2) < \text{end}(x_1) < \text{end}(y_2)$ and $\text{start}(y_2) < \text{end}(x_2) < \text{end}(y_2) \leq \text{end}(x_1)$. Hence $\text{end}(y_2) < \text{end}(y_1)$, therefore $y_1 < y_2$. \square

For a node $x \in \text{nodes}(d_i)$ we define the *test nodes set* of x for the *preceding-overlapping* axis \mathcal{X} in d_j as $\hat{\mathcal{X}}(x, d_j) = \text{ancestor-or-self}(\text{parent}(d_j, \text{first-leaf}(x_1))) \cap \{z \in d_j \mid \text{start}(z) \leq \text{start}(x) < \text{end}(z) < \text{end}(x)\}$.

THEOREM 7. *Let \mathcal{X} denote the preceding-overlapping axis and \mathcal{A} denote the ancestor-or-self axis.*

1. $\mathcal{X}(x_1, d_j) \subseteq \hat{\mathcal{X}}(x_1, d_j)$.
2. If $\text{start}(x_1) = \text{start}(x_2) < \text{end}(x_2) \leq \text{end}(x_1)$ then $\hat{\mathcal{X}}(x_1, d_j) \supseteq \hat{\mathcal{X}}(x_2, d_j)$.
3. If $\text{end}(x_1) \leq \text{start}(x_2)$ then $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.
4. If $\text{start}(x_1) < \text{start}(x_2) < \text{end}(x_2) \leq \text{end}(x_1)$ then $\forall y_1 \in \mathcal{X}(x_1, d_j), \forall y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j) : y_1 < y_2$. Moreover, if $\mathcal{X}(x_2, d_j) = \emptyset$ then $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.

PROOF. (1) and (2) follow from using dual arguments as in the proof of Theorem 6.

(3) Let $y_1 \in \hat{\mathcal{X}}(x_1, d_j)$. We have $\text{start}(y_1) \leq \text{start}(x_1) < \text{end}(y_1) < \text{end}(x_1)$ and therefore $\text{end}(y_1) < \text{end}(x_1) \leq \text{start}(x_2)$. Hence $y_1 \notin \hat{\mathcal{X}}(x_2, d_j)$. It follows that $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.

(4) Let $y_1 \in \mathcal{X}(x_1, d_j), y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j)$. We

have that $\text{start}(y_1) < \text{start}(x_1)$ and $\text{start}(x_1) \leq \text{start}(y_2) < \text{start}(x_2)$. Hence $y_1 < y_2$.

Let $z \in \hat{\mathcal{X}}(x_2, d_j)$. If $\mathcal{X}(x_2, d_j) = \emptyset$ then $\text{start}(z) = \text{start}(x_2) < \text{end}(z) < \text{end}(x_2)$. Hence $\text{start}(x_1) < \text{start}(z)$ therefore $z \notin \hat{\mathcal{X}}(x_1, d_j)$. It follows that $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$. \square

4. ALGORITHMS FOR EVALUATION OF XPATH EXTENDED AXES OVER CONCURRENT XML MARKUP

Polynomial time evaluation algorithms for XPath queries, using DOM representation of an XML Document, are given in [11, 9]. An algorithm that evaluates XPath axes in linear time (in the size of nodes in the input XML document) is also given in [9].

In this section we present linear time complexity algorithms for evaluation of *xancestor*, *xdescendant*, *following-overlapping*, and *preceding-overlapping* for a set of nodes in a distributed XML document. As stated in theorems 4 and 5, *xfollowing* and *xpreceding* can be computed in a straightforward manner using the algorithms for efficient evaluations of the *following* and *preceding* axes ([9] gives linear time algorithms for this). Finally, *xancestor-or-self*, *xdescendant-or-self*, *overlapping*, *xancestor-or-overlapping*, and *xdescendant-or-overlapping* axes are derivatives of the other six, and their computation a simple set union operation.

Figure 6 shows the pseudocode of the four algorithms for evaluation of *xancestor*, *xdescendant*, *following-overlapping*, and *preceding-overlapping* axes, and a wrapper algorithm evaluating the axes on a node set. Each of the algorithms 1, 2, 3, and 4 takes as the input a set of nodes $N_i \subseteq \text{nodes}(d_i)$, in increasing order in d_i , and a *target document* d_j where the output set of nodes N'_j is to be evaluated. In each algorithm pseudocode we use keywords like *isxancestor*, *isxdescendant* etc., as shortcuts of the tests in Definition 2 for the respective axis (*xancestor*, *xdescendant*, etc.).

Algorithm 1 evaluates the *xancestor* axis. The algorithm selects first only the nodes that start at distinct positions (lines 5-9), as nodes starting at the same position have overlapping *xancestors* (Theorem 2(3 and 4)). Then the *xancestor* nodes are added to the output set of nodes as each node in *target document* d_j is visited. Each node in N_i and each node in d_j is visited once. The correctness of this algorithm follows from Theorem 2.

Algorithm 2 evaluates the set of *xdescendant* nodes in d_j for each node in the input set N_i . The algorithm checks for each node $y \in \text{nodes}(d_j)$ whether or not y is an *xdescendant* of a node in N_i . Each node in N_i , as well as each node in $\text{nodes}(d_j)$ is visited only once. The correctness of the algorithm follows from Theorem 3. Nodes in N_i and nodes in d_j are visited in their document order, based on the order preserving properties in Theorem 3(3,4). Special treatment for nodes without text node descendants (*start* and *end* tags are at the same position) is implemented in lines 5-7.

Algorithm 3 computes the *following-overlapping* set of nodes for an input set $N_i \subseteq d_i$ and a *target document* d_j . For each node x in the input set, all *test nodes* of x in d_j are examined. The correctness of the algorithm follows from Theorem 6. The algorithm uses a stack mechanism (lines 6-18) to ensure that for any node $x \in N_i$, the descendants of x that have the same *end tag* position as x are skipped (cf. Theorem 6(2), these nodes have the same *test nodes set* as x). As it follows from Theorem 6(4) it might be the case that

Algorithm 1: xancestor evaluation

```

XANCESTOR( $N_i, d_j$ )
(1)  $N'_j = \emptyset$ 
(2)  $x \leftarrow$  first node in  $N_i$ 
(3)  $y \leftarrow$  root( $d_j$ )
(4) while  $y \neq NIL \wedge x \neq NIL$ 
(5)   while next node in  $N_i$  starts at
      start( $x$ )
(6)     if start( $x$ )  $\neq$  end( $x$ )
(7)        $x \leftarrow$  next node in  $N_i$ 
(8)     else
(9)       next node in  $N_i$ 
(10)    if  $y$  isxancestor of  $x$ 
(11)      append  $y$  to  $N'_j$ 
(12)       $y \leftarrow$  next node in  $d_j$ 
(13)    else if start( $x$ )  $<$  start( $y$ )  $\vee$  start( $x$ )  $<$ 
      end( $y$ )  $<$  end( $x$ )
(14)       $x \leftarrow$  next node in  $N_i$ 
(15)    else
(16)       $y \leftarrow$  next node in  $d_j$ 
(17)  return  $N'_j$ 

```

Algorithm 3: following-overlapping evaluation

```

following-overlapping( $N_i, d_j$ )
(1)  $N'_j = \emptyset$ 
(2) stack  $S =$  empty
(3)  $index \leftarrow 0$ 
(4)  $S.push(index)$ 
(5) foreach  $x \in N_i, start(x) < end(x)$ 
(6)   if end( $x$ ) =  $index$ 
(7)     continue
(8)   else if end( $x$ )  $>$   $index$ 
(9)      $index \leftarrow end(x)$ 
(10)     $i \leftarrow S.peek()$ 
(11)    if  $index > i$ 
(12)       $S.pop()$ 
(13)    else if  $index = i$ 
(14)       $S.pop()$ 
(15)    continue
(16)   else
(17)      $S.push(index)$ 
(18)      $index = end(x)$ 
(19)   foreach  $y \in \mathcal{X}(x, d_j)$ 
(20)     if  $y$  isfollowing-overlapping  $x$ 
(21)       if  $y \in N'_j$ 
(22)         break
(23)       append  $y$  to  $N'_j$ 
(24) return  $N'_j$ 

```

Algorithm 2: xdescendant evaluation

```

XDESCENDANT( $N_i, d_j$ )
(1)  $N'_j = \emptyset$ 
(2)  $x \leftarrow$  first node in  $N_i$ 
(3)  $y \leftarrow$  root( $d_j$ )
(4) while  $y \neq NIL \wedge x \neq NIL$ 
(5)   if start( $x$ ) = end( $x$ ) and the next node
      in  $N_i$  starts at start( $x$ )
(6)      $x \leftarrow$  next node in  $N_i$ 
(7)     continue
(8)   if  $y$  isxdescendant of  $x$ 
(9)     append  $y$  to  $N'_j$ 
(10)     $y \leftarrow$  next node in  $d_j$ 
(11)   else if end( $x$ )  $<$  start( $y$ )
(12)      $x \leftarrow$  next node in  $N_i$ 
(13)   else
(14)      $y \leftarrow$  next node in  $d_j$ 
(15) return  $N'_j$ 

```

Algorithm 4: preceding-overlapping evaluation

```

following-overlapping( $N_i, d_j$ )
(1)  $N'_j = \emptyset$ 
(2)  $current-index \leftarrow 0$ 
(3) foreach  $x \in N_i, start(x) < end(x)$ 
(4)   foreach  $y \in \mathcal{X}(x, d_j)$ 
(5)     if start( $y$ ) =  $current-index$ 
(6)       continue
(7)     if  $y$  ispreceding-overlapping  $x$ 
(8)       if  $y \in N'_j$ 
(9)         break
(10)      append  $y$  to  $N'_j$ 
(11)       $current-index \leftarrow start(y)$ 
(12) return  $N'_j$ 

```

Algorithm 5: Extended XPath axis evaluation

```

Input:  $N = N_1 \cup \dots \cup N_k$ 
Output:  $N' = N'_1 \cup \dots \cup N'_k$ 
 $\mathcal{X}evaluation(N)$ 
(1) for  $j = 1$  to  $k$ 
(2)    $N'_j = \emptyset$ 
(3)   for  $i = 1$  to  $k, i \neq j$ 
(4)      $N'_j \leftarrow N'_j \cup \mathcal{X}(N_i, d_j)$ 
(5)    $N'_j \leftarrow N'_j \cup \mathcal{A}(N_i)$ 
(6) return  $N'_1 \cup \dots \cup N'_k$ 

```

Figure 6: Algorithms for evaluation of Extended XPath axes.

some of the nodes in d_j visited as *test nodes* for some node $x \in N_i$ are visited again as *test nodes* for some of the children of x . This happens if x has no *following-overlapping* nodes but some child does. However, this situation does not propagate down to x 's other descendants (since the respective child node of x has *following-overlapping nodes*). Moreover, different children of x visit different nodes in the *test nodes set* of x . Consequently, no node in d_j is visited more than two times.

Algorithm 4 evaluates *preceding-overlapping* axis for a set of nodes $N_i \in \text{nodes}(d_i)$ and a *target document* d_j , $i \neq j$. The algorithm finds the *preceding-overlapping* nodes for each node x in the input by examining the *test nodes set* for x in d_j (correctness follows from Theorem 7(1)). Similarly to Algorithm 3, there are cases (Theorem 7(4)) when some of the nodes in d_j are visited more than once. A argument mirroring the one for Algorithm 3, can easily lead to the conclusion that no node in d_j is visited more than two times.

THEOREM 8. *Algorithms 1, 2, 3, and 4 evaluate extended XPath axes for a set of nodes $N_i \subseteq \text{nodes}(d_i)$ and a target document d_j in $O(|N_i| + |\text{nodes}(d_j)|)$ time.*

THEOREM 9. *Algorithm 5 computes $\mathcal{X}(N)$, $N \subseteq \text{nodes}(D)$, in $O(k|\text{nodes}(D)|)$ time.*

We need to conclude this section by specifying how the ordered set of nodes N'_1, \dots, N'_k can be obtained as the output of each *axis* evaluation within the time complexity bounds established by Theorem 9. Note that algorithms 1-4 use ordered node-sets as input but they not necessarily produce output node-sets in document order (per each document). Since the output node-sets of a *step* evaluation are the input of the subsequent *step* we need to make sure that Algorithm 5 (Figure 6) produces ordered node-sets. This property can be easily achieved by regenerating the output node-sets N''_1, \dots, N''_k from the output N'_1, \dots, N'_k of Algorithm 5 as follows: for each document d_j , $1 \leq j \leq k$, visit all nodes in document order and output in N''_j only nodes in N'_j . This operation takes no more than $O(|\text{nodes}(D)|)$ time, hence the complexity bound in Theorem 9 still holds.

5. EXPERIMENTAL RESULTS

We have fully implemented in Java an extension of XPath language that includes all the axes described in Definition 2. We call this processor *GOXPath*. In this section we describe our preliminary study of the efficiency of this processor.

We compare the performance of *GOXPath* to the performance of two baselines. Our first baseline is the performance of Xalan XPath processor [4] on XPath queries that are equivalent to the enhanced XPath queries processed by *GOXPath*. For our second baseline we have implemented the TEI-recommended method of representing concurrent markup using fragmentation [17]. On top of DOM trees generated using this method we have built a processor evaluating the same axes as *GOXPath*. We call this processor *FragXPath*.

Three experiments have been conducted and are reported here. The goals of the experiments were: (a) to compare the execution of *GOXPath* queries with the execution of equivalent XPath queries by Xalan on a document representation that used milestones; (b) to compare the evaluation of a path expression in *GOXPath* and *FragXPath*; (c) to test

the running time of evaluating the axes from Definition 2 for *GOXPath*. The tests were run on a Dell GX240 PC with 1.4Ghz Pentium 4 processor and 256 Mb main memory. The data input for the first two experiments was a distributed XML document obtained by multiplying the XML samples shown in Figure 2. For the third experiment we randomly generated XML data as described below.

Experiment 1. For this experiment we used the query from Section 1. The Xalan processor working on the document with milestones received the following query:

```
/descendant::w[string(.)="charges"]
/preceding::sentence[1][ancestor::page[@no="1"] or
following::sentence[1][ancestor::page[@no="1"]]]
```

GOXPath was evaluated on the query:

```
/descendant::sentence[descendant::w[string(.)=
"charges"] and (xancestor::page[@no="1"] or
overlapping::page[@no="1"])]
```

The results of the experiments are shown in Figure 7 and they clearly show that *GOXPath* outperforms Xalan used for milestones. The experiment has two parts: (i) we tested the two queries on documents of increasing size (left), and (ii) we tested the queries on documents of approximately³ the same size but increased the number of hierarchies (right). In the first part, the size of the document is measured in the number of repetitions of the XML fragment from Figure 2. The graph shows that while the processing time for *GOXPath* increased linearly and slowly with the increase in the size of the document, the running time of Xalan increased drastically, showing, what appears to be quadratic dependency on the size. For the second part, we used 100 repetitions of the abovementioned fragment. We then created four extra markup hierarchies with three XML elements in each hierarchy and marked up the content with 3-4 elements from each additional hierarchy. The graph shows that increase in the number of hierarchies did not affect the running time of *GOXPath* processor, but increased the running time of Xalan.

Experiment 2. In the second experiment we have tested the query `/descendant::page/xdescendant::*` on both *GOXPath* and *FragXPath* processors: The query expresses basic relationship between features of different structures. In this case, all features (over all hierarchies) strictly contained in “page” markup are returned. The results are shown in Figure 8 (left). The size of the document was measured the same way as in Experiment 1, only now, we tested for 5000 and 10000 repetitions as well. It takes around 3 seconds for *GOXPath* to process this query over the largest document, while *FragXPath* takes just under 14 seconds.

Experiment 3. The tests for this experiment were run on randomly generated distributed XML documents with 5 components over 100,000 character content string⁴. As the inde-

³The extra nodes introduced by fragmentation to preserve well-formedness are compensated by the extra leaf nodes in *GODDAG*.

⁴We have used the same DTD repeated five times, with tags renamed in each copy. The DTD contained 10 XML elements. The XML files generated for the tests ranged from 135Kb to 1MB.

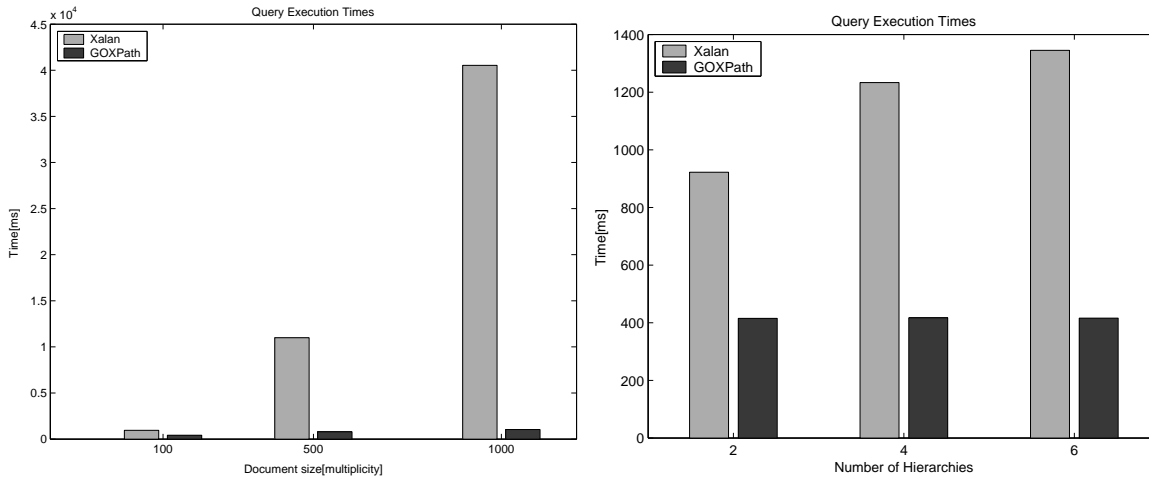


Figure 7: Searching for words.

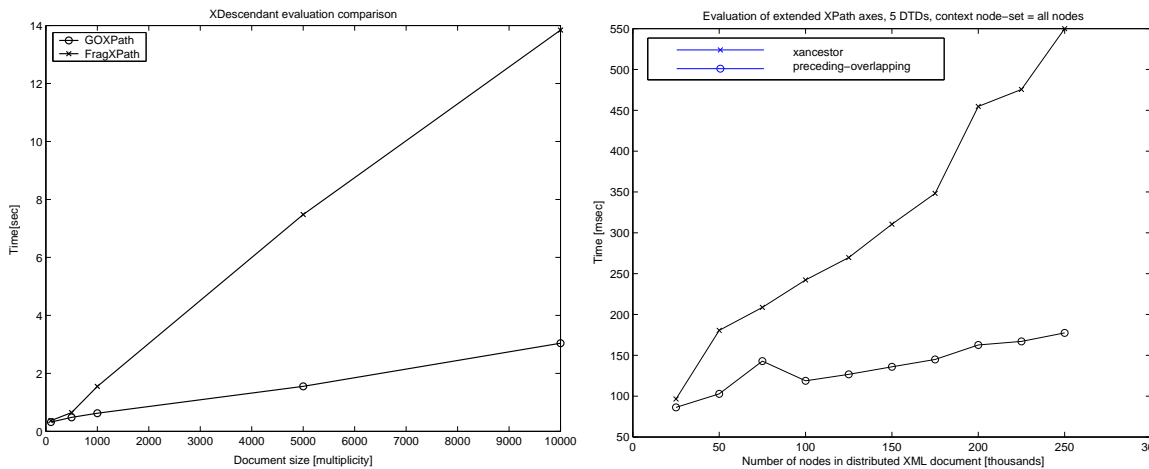


Figure 8: Evaluating *xancestor* and *preceding-overlapping*.

pendent variable we use the number of nodes in the GODDAG of the distributed XML document. For each tested value, we have generated 10 distributed XML documents and evaluated the axes *xancestor* and *preceding-overlapping* on a context node set that included *all* nodes in the GODDAG. The average times obtained in the test are reported in graph in Figure 8 (right). The results show that the time it takes to evaluate one axis is linear in the size of the GODDAG.

The experiments conducted here are preliminary and a more extensive testing is currently underway. But even these experiments show that our XPath implementation over GODDAG is efficient enough to be used in practice (and in fact, it is used as part of a larger suite of tools). The experiments also suggest, that GOXPath is, indeed, more efficient than executing queries over TEI-recommended representations of concurrent markup.

6. RELATED WORK

Related work comes from two directions: research in alternative data structures for XML representation and studies of concurrent XML and research in alternative data structures to represent XML.

Our work is similar in some respects to the work of Jagadish et al [13]. The *multi-colored trees* (MCT) data structure proposed in [13] is designed to store XML trees of different hierarchies in a *shared manner* in order to speed up tree pattern queries. We used the GODDAG data structure for the very same reason. Both [13] and this work study the semantics of XPath over the respective data structures, enhance it with functionality necessary to fully represent queries.

The key difference between [13] and our work is in the scope of XML data for which the data structures studied are applicable. MCTs of Jagadish et al. are designed with data-centric XML in mind, they allow sharing of nodes, but no overlap, and the content of individual hierarchies may (and will) be different. At the same time, GODDAG is primarily designed to store multihierarchical *document-centric* XML documents. It *allows overlap in node content*, but requires that all hierarchies share exactly the same PCDATA content.

In Sections 1 and 2 we have mentioned a number of different approaches taken by the document processing and computing in humanities researchers to representation of

concurrent XML. Text Encoding Initiative Guidelines [17] propose the use of fragmentation and milestone elements, described in detail in Section 2. Durusau and O'Donnell [8] proposed Bottom-Up Virtual Hierarchies (BUVH), a representation that encoded with XPath expressions the path to each "leaf" node in each hierarchy. They have also advocated putting all markup in a single (no longer well-formed XML) file and using lazy evaluation for actual processing [7]. Other non-XML markup languages, such as TexMECS developed by Sperberg-McQueen and Huitfeldt [12] have been also proposed. On the Computer Science side, Dekhtyar and Iacob[6] have looked at the algorithms for automatic construction and maintenance of concurrent XML documents using fragmentation, while Jarmoczyk and Moore [14] have looked at segment trees as the means of representing concurrent XML. None of these works addressed the question of querying the studied representation.

Recently W3C has released a working draft of XQuery and XPath Full-Text [5]. This proposal enhances XPath functionality with more full-text operations and Information Retrieval-style ranking queries, but it does not address directly the questions studied in this paper, namely, representation and querying of multihierarchical, overlapping markup.

7. REFERENCES

- [1] Reference removed for double-blind reviewing.
- [2] XML Path Language (XPath) (Version 1.0). <http://www.w3.org/TR/xpath>, Nov 1999.
- [3] XQuery 1.0: An XML Query Language. <http://www.w3.org/TR/xquery/>, Nov 2003.
- [4] Xalan-java version 2.6.0. <http://xml.apache.org/xalan-j/>, 2004.
- [5] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, D. McBeath, M. Rys, and J. S. (Eds.). Xquery 1.0 and xpath 2.0 full-text. <http://www.w3.org/TR/2004/WD-xquery-full-text-20040709>, 2004.
- [6] A. Dekhtyar and I. E. Iacob. A Framework for Management of Concurrent XML Markup. In *Proc. XSDM'2003, LNCS*, volume 2814, pages 311–322, 2003.
- [7] P. Durusau and M. O'Donnell. Declaring Trees: The Future of the Evolution of Markup? In *Proc. Conference on Extreme Markup Languages*, 2002.
- [8] P. Durusau and M. B. O'Donnell. Concurrent Markup for XML Documents. In *Proc. XML Europe*, 2002.
- [9] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. of VLDB 2002*, Hong Kong, 2002.
- [10] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of PODS, San Diego, CA.*, pages 179–190, June 2003.
- [11] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space efficiency. In *Proceedings of ICDE'03, Bangalore, India.*, pages 379–390, Mar 2003.
- [12] C. Huitfeldt and C. M. Sperberg-McQueen. TexMECS: An experimental markup meta-language for complex documents. <http://www.hit.uib.no/clus/mlcd/papers/texmecs.html>, February 2001.
- [13] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful xml: one hierarchy isn't enough. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 251–262. ACM Press, 2004.
- [14] J. W. Jaromczyk and N. Moore. Geometric data structures for multihierarchical xml tagging of manuscripts. In *Proc. 20th European Workshop on Computational Geometry (EWCG)*, March 2004.
- [15] M. M. Rabb. The civil rights program - letter and statement by the attorney general. The Dwight D. Eisenhower Library, Abilene, KS, http://www.eisenhower.utexas.edu/dl/Civil_Rights_Civil_Rights_Act/CivilRightsActfiles.html, April 10 1956.
- [16] A. Renear, E. Mylonas, and D. Durand. Refining our notion of what text really is: The problem of overlapping hierarchies. *Research in Humanities Computing*, 1993. N. Ide and S. Hockey, (Eds.).
- [17] C. M. Sperberg-McQueen and L. Burnard(Eds.). Guidelines for Text Encoding and Interchange (P4). <http://www.tei-c.org/P4X/index.html>, 2001. The TEI Consortium.
- [18] C. M. Sperberg-McQueen and C. Huitfeldt. GODDAG: A Data Structure for Overlapping Hierarchies. In *Principles of Digital Document Processing, DDEP/PODDP 2000, Munich*, pages 139–160, Sept. 2000. Early draft presented at the ACH-ALLC Conference in Charlottesville, June 1999.
- [19] P. Wadler. Two semantics for XPath, 1999. <http://www.cs.bell-labs.com/who/wadler/topics/xml.html>.